

# Finite Automata and Formal Languages

TMV026/DIT321 – LP4 2010

Lecture 1  
March 16th 2010

Overview of today's lecture:

- Course Organisation
- Overview of the Course
- Formal Proofs

## Course Organisation

**Lectures:** Tuesday 10–11:45 and Thursdays 13.15–15 in EL42

Ana Bove, [bove@chalmers.se](mailto:bove@chalmers.se)

**Exercise Sessions:** Mondays 13.15–15 in EL42, *VERY* important!!

Bassel Mannaa, [bassel@student.chalmers.se](mailto:bassel@student.chalmers.se)

**Assignments:** There will be non-obligatory weekly assignments which will generate bonus points valid *ONLY* on the exams during 2010.

**Exams:** May 27th and August 27th.

No books or written help during the exam.

**Level:** This course is a *bachelor* course.

## Course Organisation (Cont.)

**Book:** *Introduction to Automata Theory, Languages, and Computation*, by Hopcroft, Motwani and Ullman. Addison-Wesley.

Both second or third edition are fine.

We will cover chapters 1 to 7 and a bit of chapter 8 (if times allows).

**Wikipedia:** <http://en.wikipedia.org/wiki/>

**Web Page:** <http://www.cse.chalmers.se/edu/course/TMV026/>

Accessible from Chalmers "studieportalen".

**Course Evaluation:** I need 2 student representatives (1 GU + 1 CTH) by Thursday (next lecture).

## Automata

### Dictionary definition:

Main Entry: automaton

Function: noun

Inflected Form(s): plural automatons or automata

Etymology: Latin, from Greek, neuter of automatōs

Date: 1645

1 : a mechanism that is relatively self-operating;  
especially : robot

2 : a machine or control mechanism designed to follow  
automatically a predetermined sequence of operations or  
respond to encoded instructions

3 : an individual who acts in a mechanical fashion

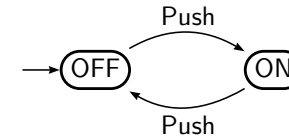
## Applications

Models for ...

- Software for designing circuits
- Lexical analyser in a compiler
- Software for finding patterns in large bodies of text such as collection of web pages
- Software for verifying systems with a finite number of different states such as protocols
- Real machines like vending machines, telephones, street lights, ...
- Application in linguistic, building of large dictionary, spell programs, search
- Application in genetics, regular pattern in the language of protein

## Example: on/off-switch

A simple non-trivial finite automaton:



*States* represented by “circles”.

One state is the *start* state, indicated with an arrow into it.

Arcs between states are labelled by observable *events*.

Often we need one or more *final* states, indicated with a double circle.



## Functional Description of on/off-switch

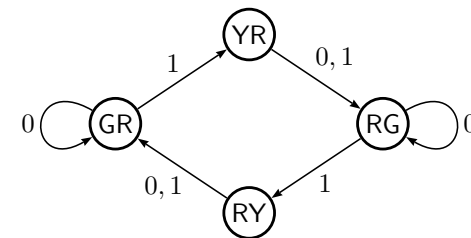
Let us define 2 functions  $f_{\text{OFF}}$  and  $f_{\text{ON}}$  representing the 2 states of the automaton.

The input can be represented as a “finite list” give by  $N = 0 \mid P N$ .

The description of the automaton is:  $f_{\text{OFF}}, f_{\text{ON}} : N \rightarrow \{0, 1\}$

$$\begin{aligned}
 f_{\text{OFF}} 0 &= 1 & f_{\text{ON}} 0 &= 0 \\
 f_{\text{OFF}} (P n) &= f_{\text{ON}} n & f_{\text{ON}} (P n) &= f_{\text{OFF}} n
 \end{aligned}$$

## More Examples of Automata: Simple Traffic Signal

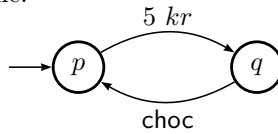


Transitions: 0 (no traffic detected), 1 (traffic detected).

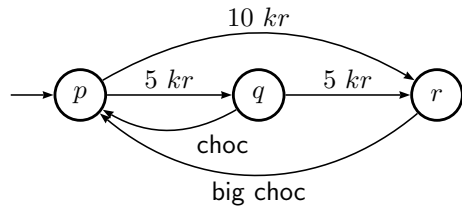
The states represent a situation: GR one traffic light is Green and the other is Red.

### Example: Vending Machines

A simple vending machine:



A more complex vending machine:



What does it happen if we ask for a chocolate on p?

State q remembers it had got 5 kr already.

### Problem: The Man, the Wolf, the Goat and the Cabbage

A man with a wolf, a goat and a cabbage is on the left bank of a river.

There is a boat large enough to carry the man and only one of the other three things. The man wish to cross everything to the right bank.

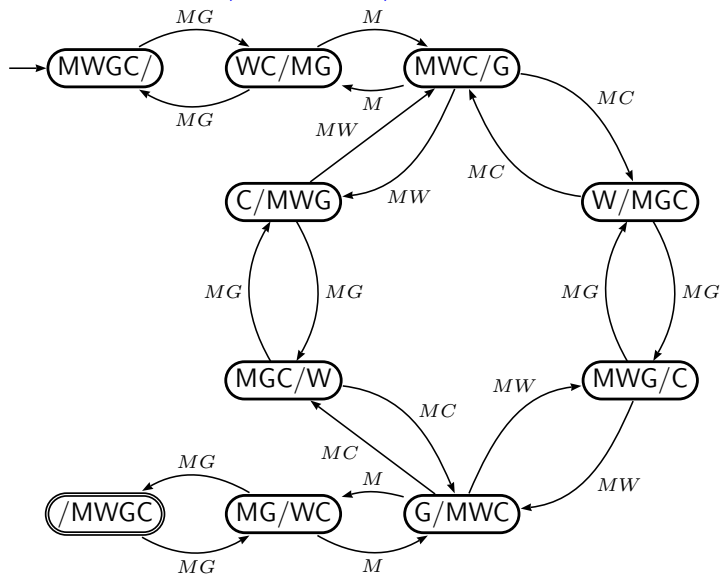
However if the man leaves the wolf and the goat unattended on either shore, the wolf surely will eat the goat.

Similarly, if the goat and the cabbage are left unattended, the goat will eat the cabbage.

**Puzzle:** Is it possible to cross the river without the goat or cabbage being eaten?

**Solution:** We write all the possible transitions, and look for possible paths between two nodes.

### Solution: The Man, the Wolf, the Goat and the Cabbage



### Overview of the Course

- Formal proofs
- Regular languages
- Context-free languages
- (Turing machines)

## Formal Proofs

Many times you will need to prove that your program is “correct” (satisfies a certain specification).

In particular, you won’t get a complex program right if you don’t understand what is going on.

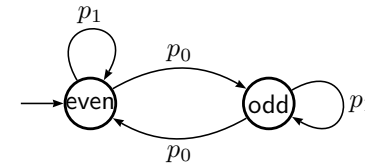
**Example:** To make an iteration/recursion correct one needs an inductive hypothesis which is consistent with the iteration/recursion.

Different kind of formal proofs:

- Deductive proofs
- Proofs by contradiction
- Proofs by counterexamples
- Proofs by (structural) induction

## Example: Parity Counter

The states of an automaton can be thought of as the *memory* of the machine.



Two events:  $p_0$  and  $p_1$ .

The machine do nothing on the event  $p_1$ .

The machine counts the parity of the number of  $p_0$ 's.

A finite-state automaton has *finite memory!*

## Functional Description: Parity Counter

Let us define 2 functions  $f_{\text{even}}$  and  $f_{\text{odd}}$  representing the 2 states of the automaton.

The input can be represented by the data type  $T = 0 \mid p_0 T \mid p_1 T$ .

The description of the automaton is:  $f_{\text{even}}, f_{\text{odd}} : T \rightarrow \{0, 1\}$

$$\begin{array}{ll}
 f_{\text{even}} 0 = 1 & f_{\text{odd}} 0 = 0 \\
 f_{\text{even}} (p_0 n) = f_{\text{odd}} n & f_{\text{odd}} (p_0 n) = f_{\text{even}} n \\
 f_{\text{even}} (p_1 n) = f_{\text{even}} n & f_{\text{odd}} (p_1 n) = f_{\text{odd}} n
 \end{array}$$

We now would like to prove that  $f_{\text{even}} n = 1$  iff  $n$  contains an even number of constructors  $p_0$ .

## Example: on/off-switch

Recall the description of the on/off-switch.

We would like to prove that:

the automaton is in state OFF after  $n$  pushes iff  $n$  is even

and

the automaton is in state ON after  $n$  pushes iff  $n$  is odd.

Alternatively, we could prove that:

$f_{\text{OFF}} n = 1$  iff  $n$  is even

and

$f_{\text{ON}} n = 0$  iff  $n$  is odd.

## Regular Languages

*Finite automata* were originally proposed in the 1940's as models of neural networks.

Turned out to have many other applications!

In the 1950s, mathematician Stephen Kleene described these models using mathematical notation (*regular expressions*, 1956).

Ken Thompson used the notion of regular expressions introduced by Kleene in the UNIX system.

Both formalisms define the *regular languages*.

## Context-Free Languages

We can give a bit more power to finite automata by adding a stack that contains data.

In this way we extend finite automata into a *push down automata*.

In the mid-1950s by Noam Chomsky developed the *context-free grammars*.

Context-free grammars play a central role in description and design of programming languages and compilers.

Both formalisms define the *context-free languages*.

## Church-Turing Thesis

In the 1930's there has been quite a lot of work about the nature of *effectively computable (calculable) functions*:

- Recursive functions by Stephen Kleene
- $\lambda$ -calculus by Alonzo Church
- Turing machines by Alan Turing

The three computational processes were shown to be equivalent by Church, Kleene, (John Barkley) Rosser (1934-6) and Alan Turing (1936-7).

The *Church-Turing thesis* states that if an algorithm (a procedure that terminates) exists then, there is an equivalent Turing machine, recursively-definable function, or a definable  $\lambda$ -function for that algorithm.

## Turing Machine (ca 1936–7)

Simple theoretical device that manipulates symbols contained on a strip of tape.

It is as “powerful” as the computers we know today (in term of what they can compute).

It allows the study of *decidability*: what can or cannot be done by a computer (*halting* problem).

*Computability* vs *complexity* theory: we should distinguish between what can or cannot be done by a computer, and the inherent difficulty of the problem (*tractable* (polynomial)/*intractable* (NP-hard) problems).

## How Formal a Proof Should Be?

- Should be convincing!
- Should not leave too much out
- The validity of each step should be easily understood

Valid steps are for example:

- Reduction to definition:  
“ $x$  is a positive integer” is equivalent to “ $x > 0$ ”
- Use of hypothesis
- Combining previous facts and known statements:  
“Given  $A \Rightarrow B$  and  $A$  we can conclude  $B$  by *modus ponens*”

## Form of Statements

Statements we want to prove are usually of the form

If  $H_1$  and  $H_2 \dots$  and  $H_n$  then  $C_1$  and  $\dots$  and  $C_m$

for  $n \geq 0$

or

$P_1$  and  $\dots P_k$  iff  $Q_1$  and  $\dots Q_m$

**Note:** Observe that one proves the *conclusion* assuming the validity of the *hypotheses*!

## Proofs by Contradiction

If  $H$  then  $C$

is logically equivalent to

$H$  and not  $C$  implies “something known to be false”

**Example:** If  $x \neq 0$  then  $x^2 \neq 0$ .

## Proofs by Contrapositive

“If  $H$  then  $C$ ” is logically equivalent to “If not  $C$  then not  $H$ ”

## Proofs by Counterexample

We find an example that “breaks” what we want to prove.

**Example:** All natural numbers are odd.

## Proofs by Induction

How to prove an statement over some inductively defined object?

**Mathematical induction:** When we want to prove a property  $P$  over *all* natural numbers.

Given  $P(0)$  and  $\forall n \in \mathbb{N}. P(n) \Rightarrow P(n+1)$  then  $\forall n \in \mathbb{N}. P(n)$ .

More generally: given  $P(i), P(i+1), \dots, P(j)$  for  $j > i$ , and  $\forall n \geq i. P(n) \Rightarrow P(n+1)$  then  $\forall n \geq i. P(n)$ .

**Course of value induction:** Variant of mathematical induction.

Given  $P(i), P(i+1), \dots, P(j)$  for  $j > i$  and  $\forall i \leq m < n. P(m) \Rightarrow P(n)$  then  $\forall n \geq i. P(n)$ .

**Structural induction:** Generalisation of mathematical induction to other recursively defined object such as lists, trees,  $\dots$

**VERY** useful in computer science!

## Example: Proof by Induction

**Proposition:** Let  $f(0) = 0$  and  $f(n + 1) = f(n) + n + 1$ . Then,  $\forall n \in \mathbb{N}$ , we have  $f(n) = n(n + 1)/2$ .

**Proof:** By induction on  $n$  where  $P(n)$  is  $f(n) = n(n + 1)/2$ .

We prove that  $P(0)$  holds.

We prove that for a given  $n$  if  $P(n)$  holds then  $P(n + 1)$  also holds.

Now we have established that for *all*  $n$ ,  $P(n)$  is true!

In particular we know that  $P(0), P(1), P(2), \dots, P(15), \dots$  hold.